链滴

SpringBoot 开发秘籍 - 事件异步处理

作者：jianzh5

原文链接：https://ld246.com/article/1600095988462

来源网站：链滴

在项目实际开发过程中，我们有很多这样的业务场景：一个事务中处理完一个业务逻辑后需要跟着处另外一个业务逻辑，伪码大致如下：

```
@Service
public class ProductServiceImpl {
    ...
    public void saveProduct(Product product) {
        productMapper.saveOrder(product);
        notifyService.notify(product);
    }
    ...
}
```

很简单并且很常见的一段业务逻辑：首先将产品先保存数据库，然后发送通知。

某一天你们可能需要把新增的产品存到Es中，这时候也需要代码可能变成这样：

```
@Service
public class ProductServiceImpl {
    ...
    public void saveProduct(Product product) {
        productMapper.saveProduct(product);
        esService.saveProduct(product)
        notifyService.notify(product);
    }
    ...
}
```

随着业务需求的变化，代码也需要跟着一遍遍的修改。而且还会存在另外一个问题，如果通知系统挂，那就不能再新增产品了。

对于上面这种情况非常适合引入消息中间件（消息队列）来对业务进行解耦，但并非所有的业务系统会引入消息中间件（引入会第三方架构组件会带来很大的运维成本）。

Spring提供了事件驱动机制可以帮助我们实现这一需求。

Spring事件驱动

spring事件驱动由3个部分组成

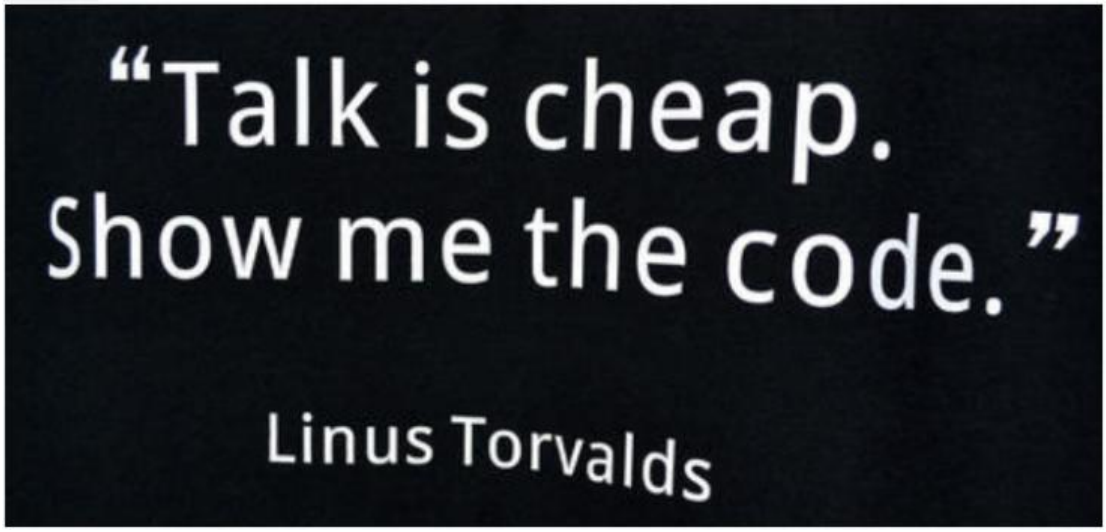
- ApplicationEvent：表示事件本身，自定义事件需要继承该类，用来定义事件
- ApplicationEventPublisher：事件发送器，主要用来发布事件
- ApplicationListener：事件监听器接口，监听类实现ApplicationListener 里onApplicationEvent法即可，也可以在方法上增加@EventListener以实现事件监听。

实现Spring事件驱动一般只需要三步：

1. 自定义需要发布的事件类，需要继承ApplicationEvent类
2. 使用ApplicationEventPublisher来发布自定义事件
3. 使用@EventListener来监听事件

这里需要特别注意一点，默认情况下事件是同步的。即事件被publish后会等待Listener的处理。如

发布事件处的业务存在事务，监听器处理也会在相同的事务中。如果需要异步处理事件，可以onApplicationEvent方法上加@Aync支持异步或在有@EventListener的注解方法上加上@Aync。



源码实战

- 创建事件

```
public class ProductEvent extends ApplicationEvent {
    public ProductEvent(Product product) {
        super(product);
    }
}
```

- 发布事件

```
@Service
public class ProductServiceImpl implements IproductService {
    ...
    @Autowired
    private ApplicationEventPublisher publisher;

    @Override
    @Transactional(rollbackFor = Exception.class)
    public void saveProduct(Product product) {
        productMapper.saveProduct(product);
        //事件发布
        publisher.publishEvent(product);
    }
    ...
}
```

- 事件监听

```
@Slf4j
@AllArgsConstructor
public class ProductListener {

    private final NotifyService notifyServcie;
```

```
    @Async
    @Order
    @EventListener(ProductEvent.class)
    public void notify(ProductEvent event) {
        Product product = (Product) event.getSource();
        notifyServcie.notify(product, "product");
    }
}
```

- 在SpringBoot启动类上增加 @EnableAsync 注解

```
@Slf4j
@EnableSwagger2
@SpringBootApplication
@EnableAsync
public class ApplicationBootstrap {
...
}
```

- 使用了Async后会使用默认的线程池SimpleAsyncTaskExecutor，一般我们会在项目中自定义一个程池。

```
@Configuration
public class ExecutorConfig {
    /** 核心线程数 */
    private int corePoolSize = 10;
    /** 最大线程数  */
    private int maxPoolSize = 50;
    /** 队列大小  */
    private int queueCapacity = 10;
    /** 线程最大空闲时间   */
    private int keepAliveSeconds = 150;

    @Bean("customExecutor")
    public Executor myExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(corePoolSize);
        executor.setMaxPoolSize(maxPoolSize);
        executor.setQueueCapacity(queueCapacity);
        executor.setThreadNamePrefix("customExecutor-");
        executor.setKeepAliveSeconds(keepAliveSeconds);

        // rejection-policy：当pool已经达到max size的时候，如何处理新任务
        // CALLER_RUNS：不在新线程中执行任务，而是由调用者所在的线程来执行
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();
        return executor;
    }
}
```